Report Number: WUCSE-2003-6

2003-01-22

The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study

Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron

General purpose middleware has been shown to be effective in meeting diverse functional requirements for a wide range of distributed systems. Advanced middleware projects have also supported a single quality-of-service dimension such as real-time, fault tolerance, or small memory footprint. However, there is limited experience supporting multiple quality-of-service dimensions in mid-dleware to meet the needs of special purpose applications. Even though general purpose middleware can cover an entire spectrum of applications by supporting the union of all features required by each application, this approach breaks down for distributed real-time and embedded systems. In particular, features from one dimension such... **Read complete abstract on page 2.**

The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study

Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron

Complete Abstract:

General purpose middleware has been shown to be effective in meeting diverse functional requirements for a wide range of distributed systems. Advanced middleware projects have also supported a single quality-of-service dimension such as real-time, fault tolerance, or small memory footprint. However, there is limited experience supporting multiple quality-of-service dimensions in mid-dleware to meet the needs of special purpose applications. Even though general purpose middleware can cover an entire spectrum of applications by supporting the union of all features required by each application, this approach breaks down for distributed real-time and embedded systems. In particular, features from one dimension such as real-time may interfere with requirements for another dimension such as fault tolerance. Furthermore, the breadth of features supported may interfere with small memory footprint requirements. In this paper, we document the results of our experiences developing special purpose middleware for an emerging class of systems: networked embedded sensors. We make two contributions to the state of the art in customized middleware for distributed real-time and embedded applications. First, we demonstrate that reduced foot-print can be achieved while maintaining or even improving real-time properties. Second, we give evidence that empirical measurement using a representative application is crucial to guide selection of feature subsets from gen-eral purpose middleware.

2003-6

The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study

Authors: Subramonian, Venkita;Xing, Guoliang;Gill, Christopher D.;Cytron, Ron

January 22, 2003

Abstract: General purpose middleware has been shown to be effective in meeting diverse functional requirements for a wide range of distributed systems. Advanced middleware projects have also supported a single quality-of-service dimension such as real-time, fault tolerance, or small memory footprint. However, there is limited experience supporting multiple quality-of-service dimensions in middleware to meet the needs of special purpose applications. Even though general purpose middleware can cover an entire spectrum of applications by supporting the union of all features required by each application, this approach breaks down for distributed real-time and embedded systems. In particular, features from one dimension such as real-time may interfere with requirements for another dimension such as fault tolerance. Furthermore, the breadth of features supported may interfere with small memory footprint requirements. In this paper, we document the results of our experiences developing special purpose middleware for an emerging class of systems: networked embedded sensors. We make two contributions to the state of the art in customized middleware for distributed real-time and embedded applications. First, we demonstrate that reduced footprint can be achieved while maintaining or even improving

The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study *

Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron
{venkita,xing,cdgill,cytron}@cse.wustl.edu
Department of Computer Science and Engineering
Washington University, St.Louis,MO

Abstract

General purpose middleware has been shown to be effective in meeting diverse functional requirements for a wide range of distributed systems. Advanced middleware projects have also supported a single quality-of-service dimension such as real-time, fault tolerance, or small memory footprint. However, there is limited experience supporting multiple quality-of-service dimensions in middleware to meet the needs of special purpose applications.

Even though general purpose middleware can cover an entire spectrum of applications by supporting the union of all features required by each application, this approach breaks down for distributed real-time and embedded systems. In particular, features from one dimension such as real-time may interfere with requirements for another dimension such as fault tolerance. Furthermore, the breadth of features supported may interfere with small memory footprint requirements.

In this paper, we document the results of our experiences developing special purpose middleware for an emerging class of systems: networked embedded sensors. We make two contributions to the state of the art in customized middleware for distributed real-time and embedded applications. First, we demonstrate that reduced footprint can be achieved while maintaining or even improving real-time properties. Second, we give evidence that empirical measurement using a representative application is crucial to guide selection of feature subsets from general purpose middleware.

Keywords: Real-Time Middleware, Distributed Embedded Systems, Sensor-Actuator Networks.

*This work was funded in part by the DARPA NEST and PCES programs. This paper is submitted to RTAS 2003 in the **RESEARCH PAPER** category for purposes of evaluation by the program committee only, and is under review by Boeing prior to final publication if accepted.

1 Introduction

General purpose middleware is increasingly taking the role that operating systems held three decades ago. Middleware based on standards such as CORBA [1], EJB [2], COM [3] and Java RMI [4] now caters to the requirements of a broad range of distributed applications such as banking transactions [5, 6], on-line stock trading [7], and avionics mission computing [8]. Different kinds of general purpose middleware have thus become key enabling technologies for a variety of distributed applications.

To meet the needs of diverse applications, general purpose middleware has tended to support a *breadth* of features. In large-scale applications, *layers* of middleware have been added to provide different kinds of services [8]. However, simply adding features breaks down for certain kinds of applications. In particular, features are rarely innocuous in applications with requirements for real-time performance or small memory footprint. Instead, every feature of an application is likely to either contribute to or detract from the application in those dimensions. Therefore, careful selection of features is crucial.

As middleware is applied to a wider range of distributed real-time and embedded applications, a fundamental tension between breadth of applicability and customization to the needs of each application becomes increasingly important. To resolve this tension, special purpose middleware must address the following two design forces.

1. The middleware should be general enough that common abstractions can be re-used across different applications in the same domain.
2. We should then be able to make fine-grained modifications to tailor the middleware to the requirements of each specific application.

To balance these design forces, two approaches to developing special purpose middleware must be considered:

- **Top-down:** sub-dividing existing general purpose middleware frameworks, *e.g.*, TAO [9], or
- **Bottom-up:** composing special purpose middleware from lower level infrastructure, *e.g.*, ACE [10].

Both approaches seek to balance reuse of features with customization to application-specific requirements. The top-down approach is preferred when the number and kinds of features required are close to those offered by a general purpose middleware implementation. In this case strategies can be selected for, possibly after being added to, the general purpose middleware to fit the requirements of the application. This has been in general the approach used to create and refine features for real-time performance in TAO.

On the other hand if the number or kinds of middleware features required differs significantly from those in available general purpose middleware, as is the case with our embedded sensor-network application, then a bottom-up approach is preferable. This is based largely on the observation that in our experience lower-level infrastructure abstractions are less inter-dependent and thus more easily decoupled than higher-level ones. It is therefore easier to achieve highly customized solutions by composing middleware from primitive infrastructure elements [11, 12] than trying to extract the appropriate subset directly from a general purpose middleware implementation.

Clearly, these approaches are complementary. Given a single application with divergent requirements and an available lower-level infrastructure framework, it may be better to apply the bottom-up approach as we have done using ACE to developing a small-footprint real-time ORB middleware framework called nORB [13]. However, refactoring an existing general purpose middleware framework top-down, especially incrementally over time, allows a greater range of robust common features to migrate downward into the lower-level infrastructure without incurring excessive delay or cost for any one development effort. For example, the migration of the `ACE_CDR` classes from TAO into ACE in earlier projects allowed reuse of those performance-tested classes on the time-critical remote-invocation path in nORB.

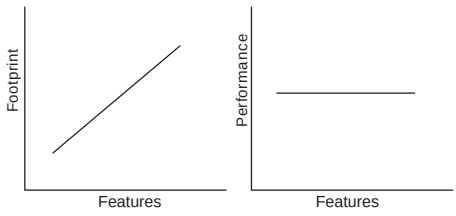


Figure 1: Features, Footprint, and Performance Goals

Figure 1 illustrates the ideal relationships between features, footprint and performance. It is reasonable to expect that there will be a reduction in footprint with fewer features. Similarly, with fewer features along the time-critical path, it seems that achieving at least comparable performance would be trivial. However, as we describe in Section 3.4, omitting a key feature may in fact *hamper* performance. Thus, reducing features for lower footprint while maintaining or improving performance in comparison to proven real-time general purpose middleware frameworks like TAO poses a significant challenge.

In this paper, we describe the challenges encountered in developing the special purpose nORB middleware framework, the solutions applied to address those challenges, and empirical comparisons of achieved performance and footprint to TAO, and several lessons learned in that process. Section 2 describes the application domain and its specific requirements for which we developed nORB. Section 3 explains how our solution achieves real-time performance comparable to TAO through careful application of design patterns, while reducing footprint by incorporating *only* those features needed by the application. In Section 4, we describe the results of experiments we conducted to examine how well nORB performs in comparison to TAO. Our middleware approach draws on previous work, which we describe in Section 5. Finally, we offer concluding remarks in Section 6.

2 Special Purpose Applications

Systems of distributed networked sensors offer an example of a special purpose application domain that exhibits the tension between design forces described in Section 1. Sensor networks are being used in a variety of different applications ranging from temperature monitoring to battlefield strategy planning [14]. Systems in this domain are characterized by the following properties: 1) highly connected networks of 2) numerous memory-constrained endsystems, with 3) stringent timeliness requirements, and 4) support for adaptive reconfiguration of computation and communication elements and their associated timeliness requirements. Sensor networks thus challenge classical approaches to distributed computing and represent an active research area with many open questions.

To identify and develop common software services

for this class of systems, DARPA's Networked Embedded Systems Technology (NEST) program [15] has supported development of different Open Experimental Platforms(OEPs), each providing its own challenge problems for a particular kind of sensor network. Our work was conducted in the context of a NEST OEP developed by Boeing. The Boeing NEST OEP seeks to achieve fine grain active control of acoustic and vibration mode damping for satellite launch vehicles and aircraft. Such active damping is made possible by a large number of Micro Electro-Mechanical Sensor (MEMS) vibration sensor-actuator nodes spread over the surface of the structure whose vibrations are to be damped. In the Boeing NEST OEP, in contrast to the wireless Berkeley motes [16], nodes are inter-connected by a wired network.

2.1 Ping Node Scheduling

To identify the current vibration mode of the structure, a *System Identification* component in the Boeing OEP sends *ping* data to sensor nodes located on the structure and identifies the mode based on the response data from the nodes. Since sensors and actuators run on limited energy resources, the number of responding nodes, called *ping nodes*, should be as small as possible and still cover the overall area to be monitored [17]. Moreover, the signaling actions of two overlapping ping nodes should be synchronized so that no interfering signals will be generated. The problem of finding a schedule for ping node responses can be solved by constraint satisfaction techniques [18]. In this paper, we use the problem of scheduling the pinging activities of sensor network nodes to compare the performance of our special purpose middleware in to that of the general purpose TAO Object Request Broker (ORB) [19].

2.2 Distributed Constraint Satisfaction

A Constraint Satisfaction Problem (CSP) [20] aims to find consistent assignments of values to a set of variables, whose inter-dependencies represent the constraints of a problem. For scalability reasons, distributed algorithms are more effective than centralized ones in large sensor networks, and it is thus desirable to apply a distributed approach to constraint satisfaction problems such as ping scheduling. In a distributed CSP, variables and constraints are distributed among multiple nodes [21]. Distributed algorithms like the *distributed breakout* [21] algorithm (DBA) and its variations [18] have been shown to be very effective for solving the distributed constraint satisfaction problem in sensor networks [18].

In particular, the ping scheduling problem can be formulated in terms of a well-known distributed CSP: *distributed graph coloring* [18]. In *distributed graph coloring*, the goal is to find a valid color assignment for all vertices of a graph, each an autonomous node in a distributed network, and the constraint being that two adjacent vertices (*i.e.*, two vertices connected by an link) cannot be assigned the same color. In the context of the ping scheduling problem, the network of sensors corresponds to a graph, a sensor-actuator node corresponds to a vertex in the graph, and connections between sensor-actuator nodes are represented by edges in the graph. The time slot scheduled for a ping node corresponds to a vertex color assignment in the distributed graph coloring problem. The example application used in our experiments described in Section 4 applies a DBA [21] to solve the *distributed graph coloring* problem, and is thus representative of sensor network CSP applications more generally. We use the term *DBA-color* for the algorithm used by our test application, as described in Section 4.

2.3 Middleware Challenges

To facilitate exchanges of local information between nodes as part of the distributed algorithm, a middleware framework that abstracts common services like communication is needed. The key challenges we faced in the design and implementation of this middleware are to:

Reuse existing infrastructure: We want to avoid developing new middleware from scratch. Rather, we want to reuse pre-built infrastructure to the extent possible.

Provide real-time assurances: Middleware itself must be predictable to allow application-level predictability.

Provide a robust DOC middleware: We chose the DOC communication paradigm over the diffusion [22] paradigm for sensor networks, since the DOC approach offers a more maintainable programming model and allows direct communication between remote nodes, which is required by the Boeing OEP.

Reduce middleware footprint: The target environment for this middleware is the sensor-actuator nodes, which have constraints on the amount of RAM/ROM on board.

3 Special Purpose Middleware Development - Our Approach

In this section we describe our approach to developing special purpose middleware, to meet the challenges described in Section 2.3. While we focus specifically on our work on nORB, the same methodology can be applied to produce middleware tailored to other special purpose applications. This section is organized as follows: Section 3.1 presents a brief survey of existing frameworks and their limitations, and describes how we leveraged existing frameworks to arrive at our solution. In Section 3.3, we describe the message formats that we use in nORB for peer-to-peer communication. Section 3.4 describes our work on critical path optimization based on previous work done in TAO. Section 3.5 summarizes how we refactored the TAO IDL compiler to build an IDL compiler for nORB. Section 3.6 briefly describes the design choices we made with respect to the lifecycle management of application specific servant objects. The concurrency strategies that we used are described in Section 3.7. Finally, in Section 3.8 we show the footprint savings that were achieved.

3.1 Middleware Solution Space

Modern software development relies heavily on re-use. Given a problem and a space of possible solutions, we try first to see whether the problem can be solved directly from an existing solution to a similar problem. If so, that solution can be applied to the problem at hand. Taking this view, we compared our problem space to a set of existing solutions, as shown in Table 1.

Challenge	Framework
Infrastructure re-use	ACE, TAO
Real-time assurances	Kokyu, TAO
Robust DOC middleware	TAO, e*ORB
Reduced middleware footprint	UIC-Core, e*ORB

Table 1: Mapping Requirements to Possible Solutions

TAO [19, 9] and e*ORB [23, 24] appeared to be the most suitable candidate solutions based on our middleware requirements. TAO is is a widely used standards-compliant ORB built using the ADAPTIVE Communication Environment (ACE) framework [25, 10]. In addition to predictable and optimized [26, 27] ORB core [28], protocol [29, 30], and dispatching [31, 32] infrastructure, TAO offers a variety of higher level services [33, 34]. e*ORB is a customized real-time and embedded ORB that offers a reduced set of features, and a corresponding reduction in footprint.

Problem: We get more and/or less than we need. Unfortunately, faithful implementation of the CORBA standard increases the number of features supported by TAO and other similar CORBA implementations and hence results in increased footprint for the application. In the case of sensor network applications, this becomes prohibitively expensive. Furthermore, the other frameworks shown in Table 1 do not, in isolation, provide complete solutions to all of the challenges described in Section 2.3.

Although ACE reduces the complexity of the programming model for writing distributed OO applications and middleware infrastructure, it does not *directly* address the challenges of real-time assurances, time-bounded adaptation, reduced footprint, or interoperation with standards-based middleware. Kokyu [35] is a low-level middleware framework built on ACE, for flexible multi-paradigm scheduling [36] and configurable dispatching of real-time operations. Thus Kokyu supplements the capabilities of a DOC middleware but cannot replace it. The UCI-Core approach supports different DOC middleware paradigms. It offers significant re-use of infrastructure, patterns, and techniques by generalizing features common to multiple DOC middleware paradigms and providing them within a minimal metaprogramming framework, thus also addressing the challenge of reducing middleware footprint. However, it is unsuited to meet the challenges described in Section 2.3 because it does not directly support real-time assurances or time-bounded adaptation of middleware QoS properties. Moreover it does not address portability across heterogeneous platforms. Finally, e*ORB provides a closed-source ORB implementation, which does not meet our requirement for re-use of infrastructure in the face of diverse requirements of special-purpose applications

Our Solution: Use a bottom-up composition approach to get *only* the features that we need. We initially considered a top-down approach as described in Section 1, to avoid creating and maintaining an open-source code base separate from TAO. However, this approach proved infeasible due to several factors. First, the degree of implementation-level inter-dependence between features in TAO made it arduous to separate them. Second, the scarcity of mature tools to assist in identifying and decoupling needed and unneeded features made it unlikely we would meet the software release schedule needed for the Boeing OEP. Third, absent better tools it was also infeasible to validate that during refactoring we had correctly retained functional and real-time properties for the large body of TAO applications deployed outside our DOC middleware research consortium.

Therefore, we ultimately took a bottom-up compositional approach [13], starting at the ACE level and re-using as much as possible from it. By building on ACE, we reduced duplication between the TAO and nORB code bases, while achieving a tractable development approach. Figure 2 illustrates our approach. The selection of features for our special purpose middleware implementation was strictly driven by the unique requirements of the application. As we show in Section 3.8, this removal of unneeded features in turn results in footprint reduction.

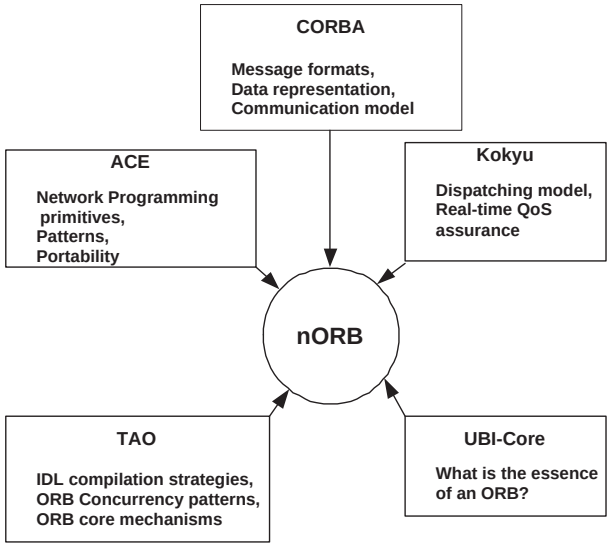


Figure 2: Re-use from Existing Frameworks

As in TAO, ACE components serve as primitive building blocks for nORB. Communication between nORB endsystems is achieved via the CORBA [1] model: the client side marshalls the parameters of a remote call into a request object and sends it to a remote server, which then demarshalls the request and calls the appropriate servant object; the reply is then marshalled into a reply object and sent back to the client, where it is demarshalled and returned to the caller. Although we did not retain strict compliance to the CORBA specification, wherever possible we have re-used concepts, interfaces, mechanisms and formats from TAO and its implementation of the CORBA standard. In the following sections, we describe the design decisions made in developing nORB and the rationale behind those decisions.

3.2 Data Types and Representation

We used the Boeing NEST OEP application domain to guide our choice of data types that nORB supports. In our DBA-color test application, for example, sequences of simple structures are exchanged between sensor-actuator nodes. nORB supports basic CORBA data types, structures and sequences. There is no support for CORBA *Any* or *TypeCode*. The support available for different data types determines the amount of marshalling and unmarshalling code that needs to be generated, which is explained further in Section 3.5. We found that the marshalling/unmarshalling code for *Any* or *TypeCode* for the DBA-color application using TAO takes about 16KB. Since the DBA-color application does not make use of these data types and it is reasonable to expect this also holds for other similar sensor network applications, we omit support for *Any* and *TypeCode* in nORB.

We use the CORBA standard Common Data Representation (CDR) as the data format on the wire in nORB. The ACE framework provides excellent support for efficient CDR marshalling and unmarshalling for a variety of data types. The ACE CDR classes were originally developed for TAO, and are extensively re-used in nORB. This in turn illustrates the benefits of pushing implementation mechanisms down from higher-level middleware such as TAO into lower level frameworks such as ACE.

3.3 Special Purpose Messaging Protocols

We need to define protocols that enable communication between end-system components, while meeting the footprint reduction challenge described in Section 2.3. Such protocols define both the sequence and format of messages exchanged between end-systems.

Previous work [37] has shown that optimizations can be achieved by the principle patterns of 1) relaxing system requirements and 2) avoiding unnecessary generality. Protocols in TAO like *GIOP-Lite* [37] are designed according to this principle. Similarly, we identify a limited subset of the types of messages supported by the CORBA specifications, so that we incur only the necessary footprint, while providing all features required by the application. The Request, Reply, Locate Request, and Locate Reply message types are supported by nORB.

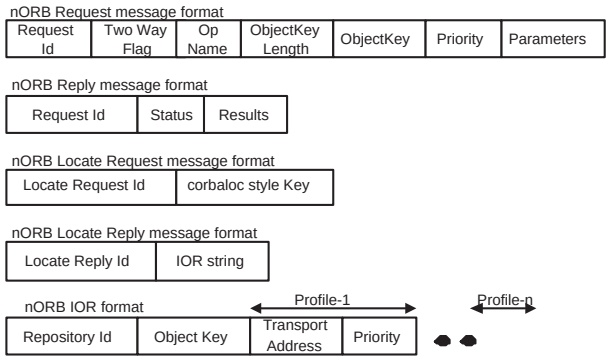


Figure 3: nORB IOR and Message Formats

Figure 3 shows the formats of different messages and of the Interoperable Object Reference (IOR) used to invoke remote methods in nORB. The format of the Request and Reply messages closely resembles that of the GIOP Request and Reply messages respectively, the major difference being the elimination of the service context field in the request and reply headers. The nORB client builds a Request message and sends it to the nORB server that sends a Reply back to the client. Based on the value of the *Two Way Flag* in the Request message, the client waits to get the Reply message from the server.

3.4 Critical Path Optimization

To achieve remote method invocation performance that is comparable with TAO, we first identified the critical path through the ORB while making a remote call. On the client side, the critical path consists of the following actions: marshalling the remote call parameters into a request message, sending the request message through a TCP socket, receiving the reply from the server, and demarshalling the return values from the reply message. Similarly on the server side, the critical path consists of the following: receiving the request from the client, demarshalling the parameter values from the request message, making the operation upcall, marshalling the call return values into a reply message, and sending the reply message through TCP socket back to the client.

Next, we took timing measurements at key checkpoints along the critical path in both nORB and TAO. The check points were 1) when the client makes a remote call, 2) when the connection to the server is established, 3) when the server receives the request, and 4) when the server dispatches the request to the remote object.

Problem: Unnecessary system calls in the critical path result in reduced performance. We found that on the server side, reading data from different client connections was unnecessarily interleaved in nORB and hence the latency between the first and the last read operation for one request was very high. Normally only one read operation is necessary to receive a request from the client. The ORB dispatches an upcall to a servant only after the whole request is read, and hence the delay incurred by multiple read operations hurts the overall performance of the server. In comparison, TAO does not exhibit this overhead and receives each request in a single read operation.

After further investigation, we traced the problem to the inefficient manner in which requests were being sent on the client side of nORB. When nORB handled a request from the client side application, two write operations were being used to send a GIOP request to the server. The data written by the second write operation could be buffered in TCP layer while the first data chunk of the request has already been delivered to the server. Thus the GIOP request was segmented and its receipt at the server spanned multiple read operations. This effect worsened when other requests came to the server before the arrival of the later chunk of a segmented GIOP request.

We also found that reading a request on the server was done using two *recv* system calls on the connection stream - first for reading the request header and the second

one for reading the request body, whereas the read operation could be done using a single *recv* call for relatively small payloads. This was done despite the request body already being present in the TCP buffer.

Solution: Reduce the number of system calls in the critical path. We achieved a single write operation on the client side by applying the well-known *gather-write* [38] technique. On the server side, we optimized reading a request so that if possible the request header and body are read using a single *recv* call. While this solution is obvious in hindsight, the problems were not detected during the initial development of nORB or even during its first deployment in the Boeing OEP, all conducted by experienced developers of advanced middleware and distributed applications. Rather, this problem first came to light during careful timing measurements performed to compare nORB and TAO in our DBA-color application. Given the myriad features within even a reduced middleware framework, it is therefore essential to perform significant performance testing along the critical path of special purpose middleware and of related general purpose middleware for comparison, *during the development process itself.*

3.5 Code Generation

A subset of the CORBA Interface Definition Language(IDL) is supported by nORB. The nORB IDL compiler generates the marshalling and unmarshalling code and hides the communication and connection management from the application. The design of the nORB IDL compiler [13] is not only based on the TAO IDL compiler, and reuses some of its parts directly. The TAO IDL compiler is modular, consisting of a front-end (FE) library, a pluggable back-end (BE) library and a top-level executive. We were able to re-use the FE library as is. For the BE library, however, more significant changes were appropriate [13]. We used a tie-based approach to generate the skeleton classes and techniques to minimize virtual functions and virtual inheritance in servant classes [13]. The skeletons thus became class templates that take the implementation classes as parameters.

3.6 Life Cycle Management of services

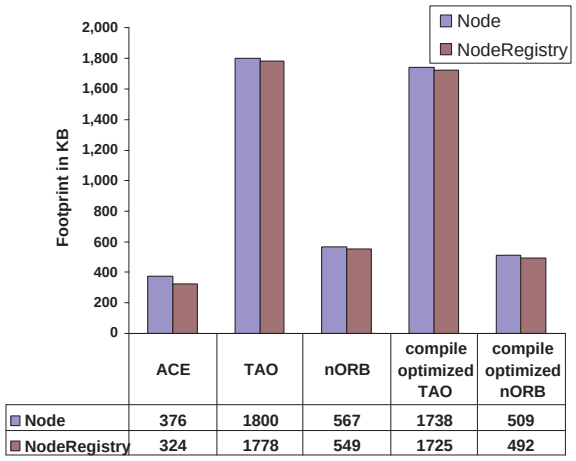
In nORB, the number of objects hosted on a node is expected to be very small, which reduces the need for a full-fledged object adapter. Servant objects are registered when the application begins to run and live as long as the duration of the application. This eliminates the need for complicated life-cycle management. Even though the resulting object adapter does not conform to the Portable Object Adapter specification, a significant amount of footprint reduction is achieved because of the reduced object adapter functionality. We have also consolidated object registration with other setup functions, by moving it from the object adapter interface to the ORB interface.

3.7 Message Flow Architecture

Our messaging and concurrency architecture is based on previous work [39, 40, 41] done in TAO. When a client makes a remote two-way function call, the caller thread needs to block until it receives a reply back from the server. The two-way function call is made on the client stub, which then marshalls the parameters into a Request and sends it to the server. The two-way function call semantics requires the caller thread to block until the reply comes back from the server. There are different strategies [39, 40, 41] to wait on the client side for the reply, of which we have chosen the *Wait on connection* strategy for implementation in nORB. On the server side, to process an incoming request and send the reply back to the client, we chose the *Direct Upcall Strategy* for nORB. With this strategy, the servant upcall is made in the context of the network I/O thread, to improve real-time predictability.

3.8 Resulting Footprint Reductions

Figure 4 shows the footprint reductions achieved by our composition approach. All measurements were taken using the *size* command. Node and NodeRegistry, described in detail in Section 4, are software components used by the DBA-color application. The application specific code in Node and NodeRegistry take about 164KB and 146KB respectively. From our footprint measurements, we found that ACE introduces an overhead of about 212KB, and unoptimized versions of TAO and nORB introduce an additional overhead of 1424KB and 1911KB respectively although compile optimization of TAO and nORB reduces the added overhead of the ORB layer to 1362KB and 133KB respectively.



	ACE	TAO	nORB	compile optimized TAO	compile optimized nORB
Node	376	1800	567	1738	509
NodeRegistry	324	1778	549	1725	492

Figure 4: Footprint Comparision for DBA-color

4 Experimental Evaluation

In this section we describe a set of experiments conducted to ensure our special purpose middleware, developed using the approach discussed in Section 3, performs well compared to the highly optimized general purpose TAO middleware for the representative DBA-color application.

4.1 Experimental Application and Platform

All experiments were conducted on a 4-machine cluster of Pentium 4 2.53GHz CPUs, each with 512MB RAM running KURT Linux 2.4.18. A NodeRegistry process opens a graph definition file and reads the topology of a graph for which we need to find a valid color assignment. In our experiments we used a 10x10 mesh of 100 Nodes, a representative sensor network topology for the Boeing OEP. Each Node represents a distributed vertex of the graph. The sequence of events is as follows:

1. NodeRegistry loads the graph from a file.
2. Nodes register with NodeRegistry.
3. NodeRegistry returns neighbor data to each Node.
4. Nodes run DBA-color until a termination condition.

A group of 25 Node processes was executed on each of the 4 machines and the NodeRegistry was executed on one of the machines. A Node communicates with its neighbors by sending *parameter messages*. There are two types of *parameter messages*: 1) Value messages, containing the current color assignment of the sending node, and 2) Improvement messages, containing the maximal reduction in conflicts that could be achieved by a color change at the sending Node.

Initially, every Node picks a random color from a color set of size equal to the diameter of the graph. The diameter of our 100-node mesh is 18. Each Node sends its current color to its neighbors. If two vertices connected by an edge have the same color, then the constraint represented by the edge is considered to be violated. After receiving individual colors from all its neighbors, each node computes the extent of such violations locally and tries to minimize such violations by searching for a different *candidate* color assignment. It then sends an *improvement*, which is a measure of the reduction in violations, to its neighbors. After receiving improvements from all its neighbors, a Node will only change its color to its *candidate* color if its own locally computed improvement is the maximum among all its neighbor Nodes. This process, called a *cycle*, is repeated until all violations are eliminated, *i.e.*, a valid color assignment is found for every Node. At this point, the algorithm is said to have *converged* and all Nodes terminate and output their final colors. Figure 5 illustrates the message interactions between a node and one of its neighbors in one cycle of the DBA-color. The original DBA algorithm is explained in detail in [21].

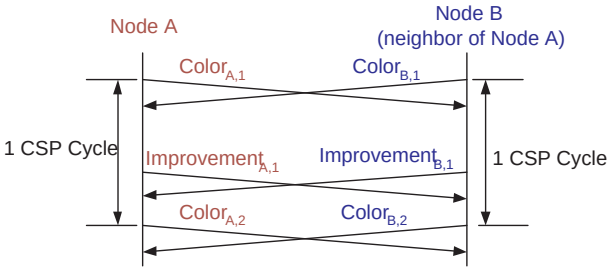


Figure 5: One DBA-color Timing Cycle

To meet the stringent performance requirements of real-time applications, we explored various optimization techniques for TAO and nORB. In particular, the following versions of nORB and TAO are used in our experiments: 1) unoptimized (default), 2) with static compiler optimizations, and 3) with runtime optimizations. For the runtime optimized versions, TAO is optimized for DBA-

color implementation via single-thread and one way function call settings, whereas nORB is optimized using the critical path techniques discussed in Section 3.4.

4.2 Performance Metrics

The following metrics were used to evaluate the performance of nORB in comparison to TAO and ACE.

Elapsed cycle times: The elapsed time for one cycle of the DBA-color algorithm is the fundamental measurement in our experiments. As mentioned in Section 4.1, a node has to wait for *parameter messages* from *all* its neighbors in each cycle of the DBA-color algorithm before it proceeds to the next cycle. Thus, a small delay in one cycle of a node will be amplified and propagated to its neighbors in the following cycles. This metric's sensitivity to delay was a major factor leading to identifying the performance variations discussed in Section 4.3.

Real-time predictability: For many soft real-time applications, the system can afford to run even with moderate numbers of failures to meet deadlines. In firm and hard real-time systems, we need to assure time bounds for higher percentages of all cases will be met. We measure time bounds on different percentiles of all samples, for strict schedulability analysis in hard real-time systems and for reliable predictability assurances in firm and soft real-time systems.

Convergence times: Convergence of the DBA-color algorithm for a given network topology is quantified by the number of cycles needed before a global solution is reached. However as noted above, variation in timing of individual algorithm steps can have a significant impact on the overall performance. Therefore, we measure the total time for the algorithm to converge over multiple repeated runs, to assess the relative overall impact of using ACE, nORB or TAO in the DBA-color algorithm.

4.3 Performance Results

We use each of the metrics described in Section 4.2 to analyze our results, as follows.

Elapsed cycle times: Figures 6 and 7 show the distribution of measured cycle times over ∼500,000 cycles of the DBA-color algorithm using ACE, nORB and TAO, up to a 50msec limit that includes 95% of all samples in each case. Measurements over 50msec are considered in the next metric. We see that each optimization is effective in

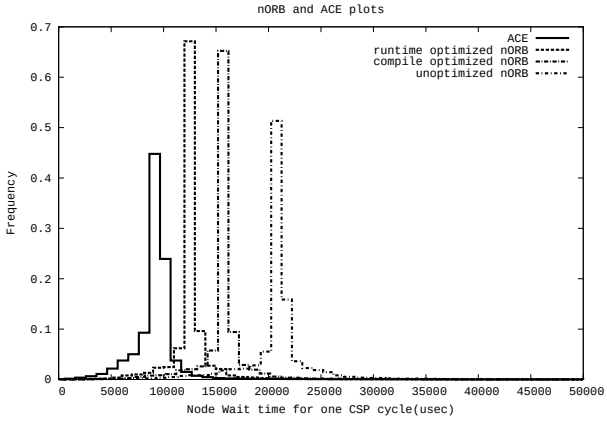


Figure 6: Performance of ACE and nORB Configurations

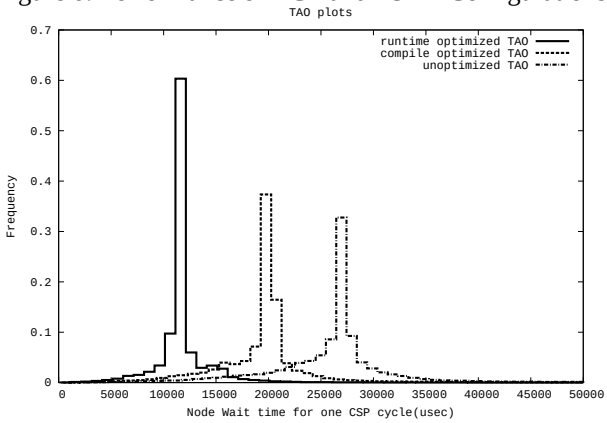


Figure 7: Performance of TAO Configurations

improving performance. Furthermore, the effects of the optimizations are more obvious for TAO, since the default configuration of TAO is aimed for multi-threaded general purpose applications. In addition, we see that one cycle of DBA-color takes similar time on average, ∼12 msec, using the runtime optimized versions of nORB or TAO with average performance marginally better for TAO.

Real-time predictability: Figures 8 and 9 show the time bounds required to assure that a given percentage of all samples will fall within that bound and thus meet a deadline at that bound. These results show that that time bounds with nORB are marginally tighter than those for TAO in the soft real-time cases from 99% to 90%, and the nORB time bounds are *significantly* tighter than those

for TAO in the firm/hard real-time cases at 99.9% and higher. We are investigating the cause of a small number of anomolous outliers for ACE, reflected in Figure 9. We suspect they are an artifact of the application or the experiment since they are associated with Node termination.

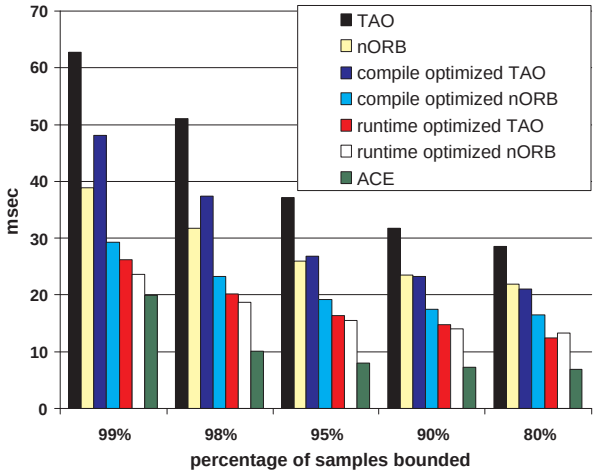


Figure 8: Lower Probability Time Bounds

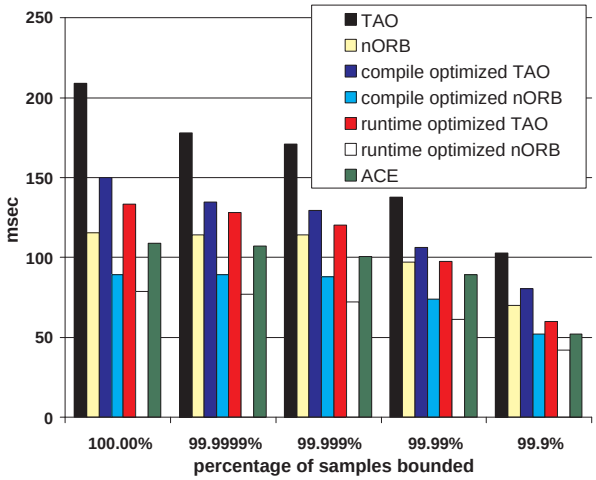


Figure 9: High Probability Time Bounds

For example, at 98% assurance, the system must be able to accommodate a delay of 21.3msec with nORB compared to 23.1msec with TAO. At 95% assurance, the system must be able to accommodate a delay of 15.9msec with nORB compared to 17.3msec with TAO.

Convergence times: Finally, Figure 10 shows the total convergence times for the DBA-color algorithm, running on ACE, TAO, and nORB. As may be expected from the

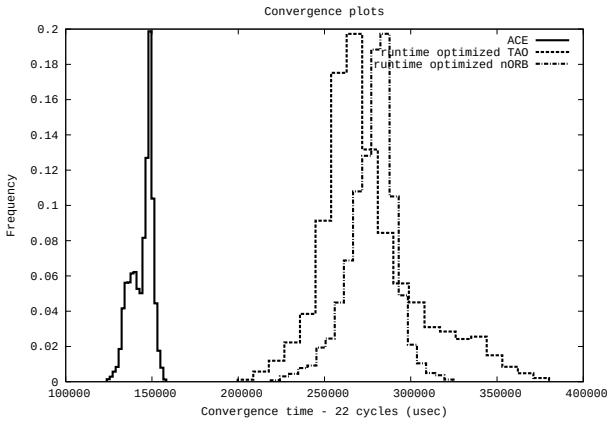


Figure 10: DBA-color Covergence on 10x10 mesh

previous cycle time and time bound figures, ACE outperforms both nORB and TAO, performing much better in both the average and worst cases. TAO slightly outperforms nORB in the average case, while nORB performs better in increasingly worse cases.

5 Related Work

In this section we describe special purpose middleware projects that address similar challenges to those described in Section 2.3.

MicroQoSCORBA: MicroQoSCORBA [42] is a middleware research project at Washington State University, focusing on middleware footprint reduction [43] through customization [44] of middleware features for deeply embedded systems. MicroQoSCORBA takes a CASE-tool approach using customized IDL compilation to generate client stubs and server skeletons supporting only the data and exception types used, choose specific transports and protocols, configure low-level marshaling properties. MicroQoSCORBA involves the system developer in selecting valid and beneficial configurations of system features. This approach thus trades increased complexity of configuring and validating the system, for greater power to tailor the details of the system.

Ubiquitous CORBA: Ubiquitous CORBA projects such as LegORB [45] and the CORBA specialization [46] of the Universally Interoperable Core (UIC) [47] focus on a metaprogramming approach to DOC middleware. Ubiquitous CORBA is similar to the ACE, Kokyu, and TAO

approaches in that it assumes a general set of primitives and a framework within which those primitives are arranged. The key difference is that the UIC contains *meta-level* abstractions that different middleware paradigms, *e.g.*, CORBA, must specialize [46], while ACE, Kokyu, and TAO are concrete *base-level* frameworks.

e*ORB: e*ORB [24] is a commercial CORBA ORB which was developed for embedded systems, especially in the Telecom domain [23]. Although the e*ORB web pages claim that e*ORB is the smallest and fastest CORBA ORB, they do not show the kinds of detailed performance and footprint comparisons with other ORBs, *in the context of a specific application*, as is presented here.

6 Conclusions

In this paper we have shown that reduced footprint can be achieved while maintaining or even improving real-time properties, in comparison to general-purpose real-time middleware. We have also illustrated the importance of careful empirical measurement within the context of a representative application, during the development of special purpose middleware. We argue that there are significant advantages to building special-purpose middleware using a bottom-up composition of lower-level infrastructure. At the same time, we note that our ability to build robust middleware using ACE depended in part on previous top-down development in TAO that then pushed key primitives, *e.g.*, the ACE CDR classes, down into ACE.

7 Acknowledgements

We gratefully acknowledge the support and guidance of the Boeing NEST OEP Principal Investigator Dr. Kirby Keller and Middleware Principal Investigator Dr. Doug Stuart. We also wish to thank Dr. Weixong Zhang at Washington University in St. Louis for providing the initial algorithm implementation used in DBA-color.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 ed., June 2002.

[2] Sun Microsystems, "Enterprise JavaBeans Specification." java.sun.com/products/ejb/docs.html, Aug. 2001.

[3] D. Rogerson, *Inside COM*. Redmond, WA: Microsoft Press, 1997.

[4] Sun Microsystems, Inc, *Java Remote Method Invocation Specification (RMI)*, Oct. 1998.

[5] L. R. David, "Online banking and electronic bill presentment payment are cost effective."

[6] K. Kang, S. Son, and J. Stankovic, "Star: Secure real-time transaction processing with timeliness guarantees," 2002.

[7] X. D'efago, K. Mazouni, and A. Schiper, "Highly available trading system: Experiments with corba," 1998.

[8] D. Corman, "WSOA-Weapon Systems Open Architecture Demonstration-Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution," in *Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 2001.

[9] Center for Distributed Object Computing, "The ACE ORB (TAO)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[10] Center for Distributed Object Computing, "The ADAPTIVE Communication Environment (ACE)." www.cs.wustl.edu/∼schmidt/ACE.html, Washington University.

[11] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," in *The OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, (Tampa Bay, FL), ACM, Oct. 2001. `http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

[12] F. Hunleth and R. K. Cytron, "Footprint and feature management using aspect-oriented programming techniques," in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pp. 38–45, ACM Press, 2002.

[13] C. Gill, V. Subramonian, J. Parsons, H.-M. Huang, S. Torri, D. Niehaus, and D. Stuart, "ORB Middleware Evolution for Networked Embedded Systems," in *Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS'03)*, (Guadalajara, Mexico), Jan. 2003.

[14] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the Physical World with Pervasive Networks," *IEEE Pervsive Computing*, vol. 1, Mar. 2002.

[15] DARPA IXO, "Networked Embedded Software Technology (NEST)." `http://www.darpa.mil/ixo/`.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 93–104, ACM Press, 2000.

[17] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *HICSS*, 2000.

[18] W. Zhang, G. Wang, and L. Wittenburg, "Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance," in *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*, 2002. to appear.

[19] D. C. S. et. al, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, Feb. 2002.

[20] S. Minton, M. D. Jonston, A. B. Philips, and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, vol. 58, pp. 161–205, 1992.

[21] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, "Distributed constraint satisfaction for formalizing distributed problem solving," in *International Conference on Distributed Computing Systems*, pp. 614–621, 1992.

[22] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Mobile Computing and Networking*, pp. 56–67, 2000.

[23] S. Aslam-Mir, "Experiences with real-time embedded CORBA in Telecom," in *OMG's First Workshop on Real-time and Embedded Distributed Object Computing*, (Falls Church, VA.), Object Management Group, July 2000.

[24] J. Garon, "Meeting Performance and QoS Requirements with Embedded CORBA," in *OMG's First Workshop On Embedded Object-based Systems*, (Santa Clara, CA.), Object Management Group, Jan. 2001.

[25] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, Apr. 1994.

[26] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[27] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, pp. 47–52, November/December 1999.

[28] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[29] A. Gokhale and D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, Jan. 1998.

[30] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[31] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, Nov. 1997.

[32] I. Pyarali, C. O'Ryan, and D. C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[33] T. H. Harrison, C. O'Ryan, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1998.

[34] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, Mar. 2001.

[35] C. D. Gill, R. Cytron, and D. C. Schmidt, "Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems," in *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.

[36] C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings Special Issue on Modeling and Design of Embedded Software*, Jan. 2003.

[37] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *IEEE Concurrency Magazine*, vol. 8, no. 1, 2000.

[38] W. R. Stevens, *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Edition*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[39] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, Apr. 1999.

[40] D. C. Schmidt, D. L. Levine, and C. Cleeland, "Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems," in *Advances in Computers*, Academic Press, 1999.

[41] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[42] David McKinnon, *et al.*, "MicroQoSCORBA." `http://microqoscorba.eecs.wsu.edu/`.

[43] A. D. McKinnon and D. Bakken and J. Shovic, "MicroQoSCORBA: A Reflective, QoS-Enabled, Configurable MicroCORBA With CASE Support," in *Proceedings of the Second Workshop on Real-time and Embedded Distributed Object Computing*, OMG, June 2001.

[44] A. D. McKinnon and O. Haugan and T. Damania and D. Bakken and J. Shovic, "MicroQoSCORBA: A QoS-Enabled, Reflective, and Configurable Middleware Framework for Embedded Systems." `http://microqoscorba.eecs.wsu.edu/MicroQoSCORBA-November2001.pdf`.

[45] M. Roman, M. D. Mickunas, F. Kon, and R. H. Campbell, "LegORB and Ubiquitous CORBA," in *Reflective Middleware Workshop*, ACM/IFIP, Apr. 2000.

[46] Manuel Roman and Roy H. Campbell and Fabio Kon, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online*, vol. 2, July 2001.

[47] Manuel Roman, "Ubicore: Universally Interoperable Core." `www.ubi-core.com/Documentation/Universally_Interoperable_Core/universal%ly_interoperable_core.html`.